REVIEWING THE COMPUTATIONAL PERFORMANCE OF STRUCTURED AND UNSTRUCTURED GRID DETERMINISTIC S_N TRANSPORT SWEEPS ON MANY-CORE ARCHITECTURES

Tom Deakin and Simon McIntosh-Smith
University of Bristol
Merchant Venturers Building, Woodland Road, Bristol, UK
tom.deakin@bristol.ac.uk; S.McIntosh-Smith@bristol.ac.uk

Justin Lovegrove, Richard Smedley-Stevenson and Andrew Hagues
Atomic Weapons Establishment
Aldermaston, UK
Justin.Lovegrove@awe.co.uk, Richard.Smedley-Stevenson@awe.co.uk, Andrew.Hagues@awe.co.uk

ABSTRACT

In recent years the computer processors underpinning the large, distributed, workhorse computers used to solve the Boltzmann transport equation have become ever more parallel and diverse. Traditional CPU architectures have increased in core count, reduced in clock speed and gained a deep memory hierarchy. Multiple processor vendors offer a collectively diverse range of both CPUs and GPUs, with the architectures used in the fastest machines in the world ever growing in diversity of many-core architectures. Going forward, the landscape of processor technologies will require our codes to function well across multiple architectures. This ever increasing range of architectures represents a unique challenge for solving the Boltzmann equation using deterministic methods in particular, and so it is important to characterise the performance of those key algorithms across the processor spectrum.
The solution of the transport equation is computationally expensive, and so we require well optimised and highly parallel solver implementations in order to solve interesting problems quickly. In this work we explore the performance profiles of deterministic S_N transport sweeps for both 3D structured (Cartesian) and unstructured (hexahedral) meshes. The study focuses on the characteristics of computational performance which are responsible for the actual performance of a transport solver.

Key Words: sweeps, performance, processors, GPUs, parallelism.

1. INTRODUCTION

Solving the Boltzmann transport equation using deterministic methods involves a significant amount of computational resources. Computing and storing the six-dimensional angular flux in memory requires suitable numerical solver methods and a highly parallel, well optimised implementation targeting modern supercomputer systems. Supercomputers are designed with nodes consisting of one or more high performance processors, with nodes connected together using a specialised network. Since the power improvements observed by Dennard scaling have slowed [20], Central Processing Units (CPUs) have evolved to be constructed from both more complex core architectures *and* multiple such cores, with the number of cores per CPU socket growing with each new architecture release. The cores themselves also contain vector processors,

allowing the computation of multiple values simultaneously under a Single Instruction Multiple Data (SIMD) paradigm. The hierarchy of memory is also growing more complex, with multiple levels of data caches improving the latency from off-processor main memory in an opaque manner. The properties of the caches and vector processors is often unique to each processor design, and the performance properties of the cache are, as we will show in this paper, crucial to a performant transport solver. Multiple vendors therefore offer a highly diverse range of CPU processors, all of which are used in the largest supercomputers in production use today (collated in the Top 500 list [19]). In order to achieve the highest tier of performance, supercomputers have turned to accelerators most recently in their use of General Purpose Graphics Processing Units (GPGPUs, or GPUs) with half of the top 10 supercomputers worldwide using GPU technology according to the Top 500 list. Indeed, this list of the fastest machines highlights the growing diversity in many-core architectures, with the top 10 machines leveraging 8 different CPU architectures and 3 different GPU architectures between them. GPUs have embraced the latest memory technologies to offer highly improved memory bandwidths and are constructed of a large number of simple processing elements. Today, GPUs are becoming more complex and including deeper cache hierarchies and accelerators just like CPUs, and CPUs are beginning to use high bandwidth memories. This evolving range of available processor technologies will require codes to operate well across the different architectures. As such, this ever expanding range of architectures presents a unique challenge for solving the Boltzmann equation using deterministic methods in particular. It is therefore important to characterise the performance of those key algorithms across the processor spectrum.

In this work we use tried and trusted transport solver algorithms which form the cornerstone of the field. In particular we use a multigroup discretisation of the energy domain and S_N quadrature discretisation of the angular dimension. For a historical view of the computational performance of transport solvers we recommend the reader consult the review by Azmy [28]. We consider a three-dimensional (3D) spatial domain discretised on a mesh; we utilise the finite difference on structured meshes and discontinuous Galerkin finite elements on unstructured meshes. Source iterations on the scattering source form the basis of our iterative solve in order to compute both the angular and scalar flux. The solution process we study here applies a methodology commonly known as *sweeps* through the spatial domain along each angular direction in the S_N quadrature.

The execution of sweep algorithms dominates the runtime of many codes and so its performance is the subject of this paper. Indeed, in order to practically solve the equation on supercomputers, the problem domain is typically decomposed in the spatial dimension across multiple computational nodes, where each node solves a collection of cells (subdomain) of the original problem. Therefore it is important to develop understanding of both the on-node and between-node (network) performance of sweeps in order to obtain an efficient deterministic transport solver. By this we mean an implementation which makes best use of the computational resource available in order to maximise the performance. As transport is so computationally intensive, ensuring peak performance will allow for improved time to solution and present an opportunity to explore increased domain resolution or include additional physics.

We define a *kernel* to be a computational routine, in our cases the code that makes up the transport sweep. It is this key kernel that we explore, and examine its performance across a range of many-core architectures. Although the development of new transport solver algorithms and alternative sweep-free approaches is important for the future, they themselves will be re-

quired to run on diverse and highly parallel architectures. Therefore a thorough and rigorous understanding of the computational performance of the current algorithms as we present in this work is vital to ensuring viable computational approaches going forward.

In order to accelerate the effort to understand the performance of complex scientific codes on many-core architectures, applications known as *proxies* or *mini-apps* have been developed [23]. These applications are written as a simplification of a parent code, however in a manner so that they are representative of the computational performance of the physics code. This means the mini-apps capture the balance of floating-point operations to memory access, the memory access patterns and footprint, and communication and decomposition schemes. However they often run artificial problems and so are of little use to producing *physically* accurate answers. The simplicity of these applications is such that, whilst they maintain similar algorithms to the parent code, they do not have library dependencies or any links to a real physical problem. In particular, the data they use is frequently auto-generated rather than loaded from an accurate data store; indeed the numerical answers are not important, but how the algorithm itself exercises the computational hardware is the key focus. To this end they can be easily shared, optimised and researched without concern for intellectual properties and licensing restrictions.

In particular, we make the following contributions:

- We will survey the performance of transport sweeps, on both structured and unstructured meshes, across a very large selection of diverse architectures, including the latest CPUs and GPUs from a variety of vendors.
- We will investigate the performance limiting factors and describe approaches for determining the key architectural components which ultimately dictate the performance of transport sweeps at scale.

2. STRUCTURED SWEEPS

The SNAP mini-app from Los Alamos National Laboratory (LANL) was developed as a performance proxy for structured Sn transport [10]. It has the ability to generate preset test problems with unphysical data, enabling a study of the performance with no focus on solving real-world physics problems. It performs a transport sweep on a structured mesh which has been spatially decomposed via the KBA method (which is explained in the next subsection). SNAP therefore reproduces both the computational load and communications pattern of a transport physics code but without the complexity of a fully-featured code.

The **mega-sweep** application [1] is a mini-mini-application for **SNAP**, which is itself a mini-application for PARTISN (from Los Alamos National Laboratory) [7, 10]. These mini-apps seek to capture the performance profiles of transport sweep applications: SNAP of a modern transport code, and mega-sweep the performance characteristics of the sweep itself without the complexity of source iterations. The **KRIPKE** mini-application (from Lawrence Livermore National Laboratory) is an alternative proxy for structured sweeps which captures differing design decisions in writing a transport code [15]. Although ensuring timely convergence from a robust iteration scheme is important to the overall performance of a transport code, it is the

[1] `https://github.com/UK-MAC/mega-stream`

sweep itself that occurs for each of these iterations that usually contributes to the majority of the runtime. In contrast, in KRIPKE the reduction of the angular flux into the scalar flux often contributes significantly to the runtime due to a lack of data reuse as a result of the choice of implementing the operators in the Boltzmann equation. To be clear, KRIPKE does require storage of the angular flux as well as the scalar flux, but performs the moments-to-discrete and discrete-to-moments operations outside the sweep as separate and distinct computational steps. This is in contrast to SNAP where these operators are included as part of the sweep kernel itself. Focussing on the sweep itself provides a more tractable approach for performance analysis. As such, mini-apps provide agile research vehicles where it is tractable to explore the fundamental properties of the algorithm without the burdens associated with production applications. Understanding the data movement of the transport kernels is a key focus of this work.

2.1 Multi-node Scalability of Sweeps

In the case of a structured mesh a popular approach to performing a fully parallel sweep, in which the dependencies between tasks on different MPI processes are respected, is the so-called *KBA* (Koch, Baker and Alcouffe) algorithm [12]. This utilises a reduced dimensional decomposition of the mesh, comprising a stack of spatial sub-domains (also known as *pencils*), with each pencil owned by an MPI process. The length of each pencil is the same, namely the extent of the original mesh in its largest dimension. Each MPI rank completes its own local sweep, with the sweep traversing the shortest dimensions of the pencil first, such that the wave-front reaches all available MPI processes as soon as possible.

This approach does include some *build-up* time where MPI ranks are waiting for work to become available to them, and *tear-down* time where some ranks are idle having completed all their work, but others are still busy. One approach to minimising this is to perform a sweep for a single angle on the spatial subdomain, before beginning a new sweep for the next angle on that spatial subdomain [11]. This reduces the number of tasks to be completed on an MPI rank before it can communicate data to the neighbouring rank. By beginning the next sweep as soon as the first MPI rank finishes all work in the first sweep, rather than waiting for the entire sweep to complete on all MPI processes, the idle time may be further reduced. The idea of starting an angle's sweep before the preceding angle's sweep is complete is referred to as *pipelining*.

One potential optimisation of the sweep, from a network and MPI infrastructure perspective, is to delay MPI communications until there are multiple cells' flux values to send (e.g. [14]). This reduces MPI overheads, as sending fewer, larger messages takes advantage of the bandwidth of the network while reducing the orchestration time for sending and receiving individual messages. This may be implemented either via some form of message buffer, which send only when the buffer fills, or by dividing the KBA pencil into *chunks* and communicating only when all cells on a chunk have been solved for. This does stagger the wave-front's propagation across processors and increases the delay before any processor may start work. However the reduction in MPI overheads can again lead to overall performance improvements.

In a Cartesian geometry with no reflective boundary conditions, such that no coupling occurs between octants in the quadrature set, a *hybrid KBA* approach may be employed, where each pencil is further divided across two processors [24]. This has the benefit that the two octants

which would ordinarily begin on a single MPI process, may now be started simultaneously as their starting cells now lie on different processes. Octant pipelining can lead to *collisions* in which two octants' sweeps reach the same MPI process at the same time, and the process must prioritise work on one or the other of them. The impact of this has been studied both via performance modelling and numerical experiment for different spatial decompositions [25]. Collision-avoiding algorithms show better performance at small scale, but the reduced build-up and tear-down time improves the scaling of the KBA algorithm such that the performance impact of collisions is negated for large process counts.

Developing highly parallel sweep algorithms has been an important research area in order to run on IBM BlueGene/Q and (predecessor) systems (e.g. [24]). This research focusses on scheduling and aggregation of the fine-grained tasks defined to be the solution of one angular flux unknown (that is the angular flux for a single energy group for one angular direction in one spatial cell). The scheduling prioritises the sending of tasks between parallel processors so that all processes are idle for the least amount of time. A fundamental assumption in this approach is that the processors operate in a serial manner (a single instruction stream, single data stream (SISD) processer under Flynn's taxonomy of parallel processors). Therefore, the schedule is designed around making choices which do not retard the sweeps [25]. The performance models were built around these fine-grained tasks, numerically represented with a *grind time* capturing the time required to solve this one unknown. The model also included a number of network performance parameters.

We extended these performance models in order to explore the scaling on large GPU systems [3]. On GPUs however modelling the computation cost of the aggregated tasks as the product of grind time and the number of unknowns was inaccurate. For a given angular direction, a *local wavefront* follows the sweep dependency on the local spatial subdomain and exposes the maximum number of cells with satisfied dependencies as a result of total solution of the previous wavefront. We developed a model which showed high accuracy when the tasks were modelled as the product of the number of local wavefronts in each spatial chunk of the local subdomain. That is the number of angles and energy groups (which were processed all in parallel in contrast to the prior methods) was somewhat unrelated to the dominant factor in the computational cost. The performance of the GPU implementation showed speedups in line with the relative performance over large CPU systems. The scaling model shows that at high processor counts, point-to-point communication becomes the bottleneck: at 2048 GPUs 60–80% of the runtime was in communications [3]. Although the start-up/tear-down overhead of sweeps is in part responsible for this, the acceleration (decrease) of the solve time that the GPUs provided when compared to the unchanged cost of sending messages, still results in the performance being limited by message passing at scale. Our work showed that the KBA spatial decomposition still scaled well enough up to 8000 GPUs, and the earlier work of the research group at Texas A&M University shows KBA scaling to many more concurrent CPU processors (MPI ranks) [26].

2.2 On-node Parallelism

The mega-sweep code (and its parent SNAP) performs sweeps according to a KBA spatial decomposition using MPI, using the CPU vector units for updating all angles within each octant in parallel, and OpenMP threads over the energy groups under a Jacobi scheme (after [14]). Here, OpenMP worksharing directives are used on the energy group loop rather than the SPMD-

style OpenMP programming used in SNAP; this SPMD approach requires high levels of thread support from the MPI library and is not compatible with GPU (offload) architectures. The octants are swept in sequential order, and the concurrency in this domain is not exposed so that the findings may be useful to a wider range of problems where parallel octant sweeps are not available (such as problems with reflective boundary conditions). The octant sweeps are pipelined in opposite pairs to help reduce the expected start-up costs.

To best utilise GPU architectures we found that we required a concurrency scheme with more available parallelism exposed [1–3]. In addition to the concurrency in angles and groups already utilised on the CPUs, the natural spatial concurrency of cells on each wavefront plane of the *local* subdomain were included in the parallel workloads. Although the amount of parallelism is variable, a modest number of parallel cells gives sufficient parallelism to exploit all the concurrent resources of a GPU in combination with angle and group. Such parallel schemes allow for the best utilisation of the high memory bandwidth available on GPUs but are at odds with requirements for efficient sweeps as stipulated by Adams and Bailey [24]. Using LANL's SNAP mini-app, we showed our parallel scheme is limited by GPU memory bandwidth, and achieved speedups over CPUs in line with this metric [1, 2]. The developers of KRIPKE had similar findings shortly after our publication and tried additional techniques for implementation optimisation in situationally beneficial circumstances where a KBA spatial decomposition is not used [16, 18]. That optimisation is at the expense of a reduction in parallelism for the reduction of memory movement and so may not always be applicable.

2.3 Processor Performance Bounds

To this end, we were able to explore the performance limiting factors on many-core devices (CPUs and GPUs) for finite difference, structured grid, deterministic S_N transport.

In order to provide some context as to the performance of processors, we present memory bandwidth results for some high-performance processors. Triad is the classic benchmark for determining available main memory bandwidth, and codes which are main memory bandwidth bound typically align with the benchmark results [6, 17]. This kernel scales a vector array and adds it to a second vector, storing the result in a third:

$$a(i) = b(i) + \alpha \times c(i) \tag{1}$$

The throughput of memory (the memory bandwidth) can be calculated by running this kernel many times and computing the ratio of memory moved and the runtime. We use the BabelStream benchmark to calculate this for a number of CPUs and present these in Fig. 1. A higher result indicates more memory bandwidth. The results are shown normalised to the Broadwell processor which attained 131.1 GB/s. The relative performance of the transport codes will be compared to the relative performance of this benchmark; for instance the KNL processor has the highest memory bandwidth due to its use of MCDRAM high-bandwidth memory technology. Further details about the processors can be found in Table I.

As a tool to explore performance bounds of the important transport sweep solver kernel, we developed the mega-sweep mini-mini-app, extracting just the necessary computational patterns

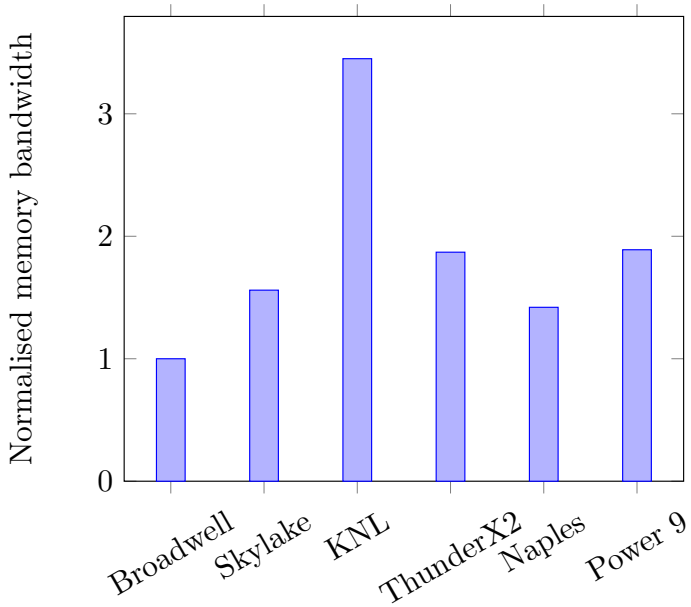

Figure 1: BabelStream Triad results for a number of multi-core CPU processors.

from SNAP so as to model the computation [4, 7]. The code does not contain all the details in order to solve the full transport equation directly, and as with most proxy application leverages synthetic data. The code uses KBA sweeps to invert the time dependent streaming operator and compute the scalar flux for a chosen number of iterations (rather than using convergence checking which would be redundant for the fake data):

$$\left(\frac{1}{v}\frac{\partial}{\partial t} + \hat{\Omega}\cdot\nabla\right)\psi(\vec{r}, E, \hat{\Omega}, t) = q \tag{2}$$

There are of course a number of simplifications and omissions here, however this captures the very heart of the kernel: the diamond difference operations and the communication and flow of data in S_N sweeps. Including additional operators and understanding how they impact the performance described here is the subject of future study.

In Fig. 2 we show some of our performance results for mega-sweep across different CPU architectures, normalised to Broadwell. Note that we present performance results for fully utilised processors and never collect or compare serial (single core) results as whole processor usage better represents real world running conditions. The non-independence of cores due to coupling in the memory subsystems in modern processors means that "per core" results are often misleading for all but the most synthetic of architectural benchmarks. The results are for a 3D spatial mesh of $80 \times 8 \times 8$ cells (recall the use of a KBA decomposition resulting in pencil shape subdomains) with 64 energy groups and 48 angles per octant. Under a traditional Roofline model, due to the low computational intensity of the sweep finite difference kernel, the performance would be classified as main memory bandwidth bound, and not bound by the rate of floating point operations (FLOP/s) [21]. It is clear by comparing the results shown in Fig. 1 with those of the sweeps in Fig. 2 that sweep computational performance does not correlate with being limited by main memory bandwidth. Indeed, processors with a high memory bandwidth do not neces-

Table I: Processor details

Processor	**Vendor SKU**	**Cores**	**Clock speed** (GHz)
Broadwell	Intel Xeon E5-2699 v4	2×22	2.2
Skylake	Intel Xeon Platinum 8176	2×28	2.1
Knights Landing (KNL)	Intel Xeon Phi 7210	64	1.3
ThunderX2	Marvell Arm	2×32	2.1 (2.5)
Naples	AMD EPYC 7551	2×32	2.0
Power 9	IBM	2×20	3.2

sarily provide fast runtimes for the sweep. This implies that neither FLOP/s or main memory bandwidth are a significant performance limiting factor for structured grid sweeps. The limiting factors are the topic of the next section (Section 2.3.1).

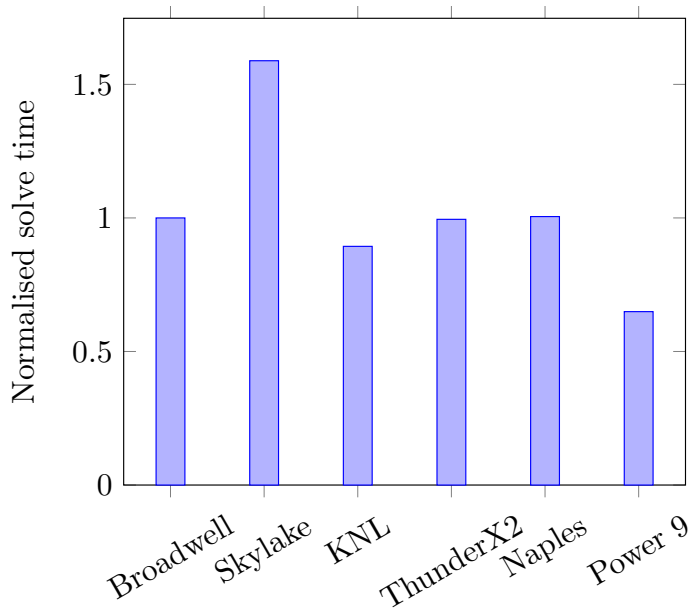

Figure 2: Performance of mega-sweep on a variety of many-core CPUs relative to Broadwell processors.

2.3.1 Locating the performance limiting factor

The preceding mega-sweep results highlight that the computational solve required by a finite difference transport sweep (between communications to the downwind neighbours) is bound by neither the speed at which floating point operations can be performed by the processor, nor the throughput of main memory. This is very counter-intuitive as the streaming access pattern of the angular flux array through memory is exceedingly similar to the memory patterns in the Triad kernels. We present some experiments which assist in locating the performance bottlenecks due to the hardware for the different processors in this study.

The following procedure was proposed by Voysey and Glover in order to help discover performance limiting factors of a code [22]. The experiment is described in terms of a two-socket CPU system, with each CPU consisting of a number of cores. The code is run in two configurations:

1. Use all cores of *one* socket.
2. Use half the cores of *two* sockets (each).

In both configurations, the code is run on the same number of physical cores. For example on the Broadwell CPUs used in this study, each socket contains 18 cores. We run our code firstly on the 18 cores of one socket (configuration 1.), and secondly on 18 cores where nine are selected in each socket (configuration 2.).

The characteristics of the design of CPU processors renders some resources shared between cores on each socket, whilst others are replicated on each core. Floating-point performance is primarily a factor of the number of CPU cores, as is the capacity and accesses to first (and often second) level memory caches. Both main memory bandwidth and last level cache capacity and access are resources shared between all cores on a socket. Therefore the relative performance differences between the two run configurations can yield an insight into where the performance bound is located: within the resources of the core or within the shared resources of the socket. The shared resources in configuration 2. have been increased, whereas the resources associated with the cores remain the same as they are for configuration 1. If the performance improves from configuration 1. to configuration 2. then the performance bound may be associated with off-core resources (main memory bandwidth, last level cache, etc.). Indeed, the available resources are typically doubled and so there is an expectation that the performance should improve two-fold. Alternatively if the performance does not improve with the second configuration then the performance bound may be associated with the on-core resources (floating-point performance, close cache performance, etc.). It is often prudent to record the clock speeds for both experiments as some CPU designs allow for an increased clock speed when not all cores are utilised, taking advantage of thermal and cooling factors.

The results of this experiment for mega-sweep are presented in Fig. 3 shown as the relative improvement yielded (if any) by configuration 2. compared to configuration 1. for each platform independently.

The Power 9, Naples and ThunderX2 processors all show little improvement from configuration 2. Recall this means that the extra shared resources introduced here provide no benefit, and therefore the performance bound is due to a feature of the core itself. The number of floating-point operations is very low and so the on-core resources most likely to be the performance

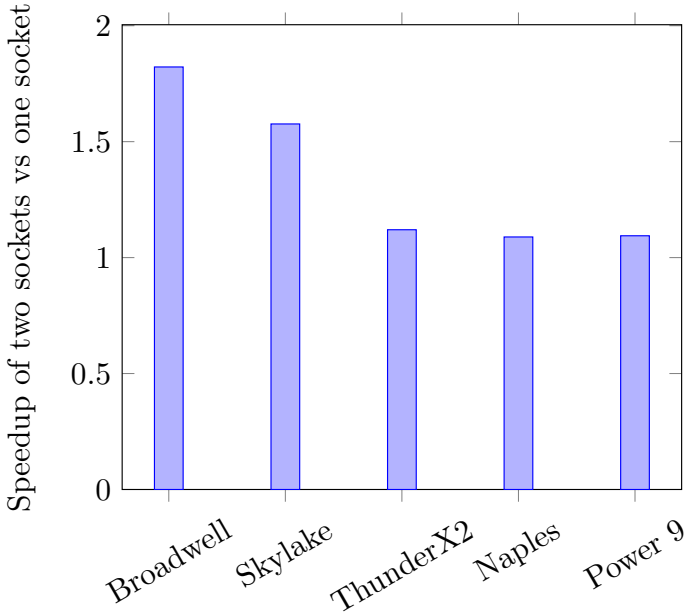

Figure 3: Relative performance improvement of mega-sweep from Configuration 1. to 2. in the Voysey experiment.

limiting factor are the caches. We will investigate this more analytically for the ThunderX2 in Section 2.3.2.

On both the Intel Xeon processors (Broadwell and Skylake) the second socket does provide some performance improvement however not as much as the two-fold improvement expected. This improvement means that on this processor the off-core factors are dominating. As in previous work ([4]) we found that non-temporal store instructions improve the performance on Intel processors by around 1.4X. For this code, such instructions cause the store of the updated cell-centred angular flux to bypass cache and be stored in main memory directly, thus providing additional cache capacity for other more reused data. It is worth noting that the clock speeds increased on the Skylake processor by 10% for configuration 2. and so the relative performance improvement should be a little diminished if the system were to be configured with a fixed clock speed. Recall from Fig. 1 and Fig. 2 that the performance improvement for both Triad and the mega-sweep problem offered by Skylake over Broadwell is 1.6X for both; thus indicating that on these class of processors main memory bandwidth is a key factor in the performance characteristics. Both these processors feature highly sophisticated memory prefetchers and along with the implementation of the non-temporal store instructions allow for highly efficient cache use and thus the bottleneck is movement of memory into the processor. Note however that the Intel KNL processor features much improved main memory bandwidth over Broadwell yet did not yield a performance improvement for sweeps (Fig. 2). As a single socket system (with a single NUMA node) we are unable to perform a Voysey style experiment, however the cache capacity on these processors is significantly reduced (per core) compared to the Xeon designs.

This experiment demonstrates that the memory hierarchy of a processor is critical to the performance of a transport sweep. It also highlights that the performance limiting factor may differ

depending on the design of the CPU, and no single bound can apply for this class of many-core architectures. However it is clear that it is the efficient use of cache that is key to obtaining good performance for the diamond difference update at the heart of the sweep.

2.3.2 Modelling runtime with cache misses

More insight into the behaviour of the processor during the execution of an application can be found by observing the performance counters embedded in the device. Such counters keep a record of a variety of events in order to capture characteristics about which parts of the processor are being exercised. Common counters record the number of cycles taken during the program's execution, the number and type of instructions executed, and memory subsystem metrics such as the number of cache accesses and whether those accesses hit or missed each level of cache.

The code was built with the Cray compiler 8.7 with the `-O3` optimisation flag along with those set by the Cray Programming Environment which specialise the compilation to the target architecture. We use the Linux `perf` tool to collect key performance counters on the Marvell ThunderX2 processor. We collect four counters during a run of mega-sweep:

1. the L1 data cache misses.
2. the total number of cycles executed.

The average latency (in cycles) can be approximated using additional performance counters:

3. a measure of the number of cycles a load in the cache miss queue is waiting to be satisfied.
4. a measure of the number of times a load enters the queue.

The ratio of these two performance counters (counters 3. and 4. above) gives an approximate latency for a cache miss for the access patten present in the program. It is important to measure this latency as the memory prefetchers will predict and satisfy a good number of the memory loads. As such, it is only those loads not in the process of being satisfied by the prefetcher or an earlier cache miss. Therefore, it is an application dependent value based on the specific memory access pattern.

We explore the two memory access patterns in the sweep kernel: forward and backward sweeps in the x-direction. For a backwards sweep, we find that there were approximately 107 billion cycles executed in the benchmark (counter 2.). Using the approximation above for the latency of a cache miss we derive a 31 cycle cost. We observed around 3.1 billion L1 cache misses (counter 1.) This results in around 96.4 billion cycles attributable to waiting for resolution of a cache miss (counter 1. multiplied by the latency of 31 as above). Specifically, 90% of the execution time (measured in cycles by counter 2.) can be estimated as waiting for memory to be loaded from the first level of cache.

A similar analysis can be performed for forward sweeps, resulting in 63% of the total cycles waiting for memory loads. This is calculated from a cache miss latency of 18 cycles, observing a similar 3.1 billion L1 cache misses (counter 1.) and an execution time of around 88.2 billion

cycles (counter 2.). In the forward case we also observe that the prefetcher is more effective which causes a greater accuracy of data preloaded into L2 cache and hence a lower miss penalty (latency) is observed for the L1 cache. This shows that for both forwards and backwards sweeps, a significant part of the total runtime comes from the time to satisfy cache misses.

The memory access pattern of a sweep is somewhat interesting due to the multi-dimensional array. The angular dimension is typically the inner-most dimension and vectorised so as to process multiple angles simultaneously. This results in an access pattern which moves along the array (in increasing address numbering) processing angles. For a sweep in a backwards direction, once all angles in a cell boundary are computed and the processing is to begin on the previous cell, a jump in memory access occurs to the start of the previous cell. This breaks the pattern of moving forwards and will likely trigger a high number of hardware prefetches to be issued, attempting to adjust to the new pattern. However this proves ineffectual as another jump occurs for the next cell and the prefetcher has caused a large amount of unused memory to be loaded into cache. Indeed, on ThunderX2 we observe a much higher number of both misses *and* prefetches generated for the backwards sweep. This shows that this congests the processing of loads and may cause eviction of useful data which was present in the cache.

One possible mitigation on those processors which do not have such a pattern known to their prefetchers is to simply recode the backwards sweeps as forwards sweeps with a simple renumbering of the angular flux arrays in memory.

As such, the performance is limited primarily by loading memory from the cache hierarchy on multi-core CPUs. This is therefore an unusual performance characterisation as many other simulation algorithms are instead limited by main memory bandwidth, and is as such an important point to discuss within the transport community.

2.4 Summary

The performance bounds of structured mesh transport sweeps are dependent on a number of factors. At scale, the communication cost (the latency of the network) is a highly dominant factor.

On CPU architectures, the memory hierarchy dictates the performance of the diamond difference kernel itself. When the problem can remain resident in cache thanks to highly performant prefetchers, the movement of memory into the processor from main memory is a key performance requirement. Otherwise, the fulfilment of memory loads from the closest caches determines the runtime of the solve. The kernel requires high data reuse of the neighbouring flux arrays whilst also streaming the very large angular flux data. It is how the cache hierarchy manages this balance of requirements that will determine the performance characteristics.

On GPU architectures on the other hand, additional concurrency in the algorithm must be exposed in order to obtain good performance. The natural independence between cells on the wavefront of the sweep provides this extra parallelism, and when combined with the concurrency in angles (within a single octant) and energy groups, sufficient parallelism is found to saturate a GPU with work [3, 9]. This extra parallelism leverages the latency hiding advantages of GPU architectures obtained by their ability to context switch quickly to hide long latency memory requests with other work. As a result, device memory bandwidth becomes the performance

limiting factor on GPU architectures.

3. UNSTRUCTURED SWEEPS

Sweeping a structured spatial mesh exposes many of the complexities of leveraging many-core processors for a highly performant sweep code. However, unstructured, high-order spatial meshes can allow for a more accurate representation of the spatial domain but require a different solution approach at the heart of the sweep. Structured meshes can be solved with upwinded finite difference, but unstructured meshes typically employ other methods. Here we use an upwinded discontinuous Galerkin finite element discretisation. The construction and inversion of the local linear system in each element (for an angle/energy group pair) becomes the kernel at the heart of the sweep.

We have developed the **UnSNAP** mini-app to explore the performance of using a 'matrix-free' discontinuous Galerkin finite element solution to the transport equation on unstructured hexahedral meshes [5, 8]. This mini-app was developed as a port of the (structured) SNAP from Los Alamos National Laboratory, and uses OpenMP and CUDA to run in parallel on CPUs and GPUs. The S_N quadrature, material data and layout, source updates and other approximations are all inherited from SNAP. The mesh is generated as an unstructured graph of three-dimensional hexahedral discontinuous Galerkin elements. The mesh allows for arbitrary order discontinuous Lagrange elements with nodes on vertices, edges, faces and within the element volume; it is also possible to represent curved and distorted elements.

It should be noted that the computational intensity of our structured grid finite difference (0.22 FLOPs/byte) and linear finite element (0.25 FLOPs/byte) discretisations are similar, as although more floating point operations occur in the finite element method, more memory reads are required [7].

UnSNAP therefore provides an ideal vehicle to focus on the on-node performance of unstructured sweeps on different many-core architectures, just as we did for structured sweeps in Section 2.. Multi-node performance is the topic of future work and will of course leverage the wealth of existing research in this area.

In order to solve the transport equation in each element efficiently, we construct the local linear system using the problem data and precomputed basis function integrals. The system is then inverted in order to compute the angular flux. The sweep dependency is followed along each angular direction, with upwinded fluxes over the boundaries used as expected. This sweep dependency is computed for each angular direction and stored in a graph.

An important metric calculated from this graph is the *t-level*. This number is the depth of the node in the graph from the root. The upwind neighbours of a particular element will have a lower t-level. In a structured mesh, this number would state that the cell had its dependencies satisfied on the t^{th} wavefront. In other words, if all elements on a given t-level are computed, then the downwind neighbours will have their dependencies satisfied for the next t-level. A simple sweep graph for one angular direction through some small unstructured mesh is shown in Fig. 4. The t-levels are numbered on along the left-hand side of the figure. Note that the t-level is an inherent property of the graphs and is not a free parameter.

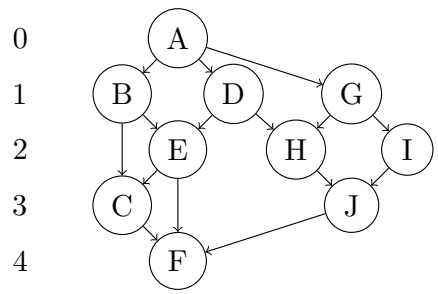

Figure 4: Illustration of t-levels in a sweep graph for a single angular direction. The arbitrarily circular nodes marked with letters represent distinct spatial finite elements. The arrows indicate the upwinded sweep dependency.

3.1 On-node Parallelism

For many-core CPU architectures, we parallelise the unstructured sweeps differently to those in the structured sweeps. We follow the sweep order graph for each angle in turn (one after the other). Each t-level in the graph contains a list of elements to solve, so we parallelise the solve for all energy groups in these elements for the particular angle using threads: this results in one solve per core. Our code is NUMA-aware as each parallel thread (assigned to a core) allocates the memory it needs for the local system; note that some shared data such as cross sections cannot be fully NUMA-aware as they are required by all cores. Solving the energy groups in parallel helps with both having sufficient parallelism and also mitigating the memory discontinuities from the unstructured nature of the grid as each core can process all energy groups for a single element, thus having a greater amount of contiguous memory access: a factor of utmost importance for performance [5]. It is perfectly possible to extend this scheme to group-dependent quadrature sets by rewriting the nested loop over the t-level and energy groups as a single loop over the total work: all groups (however many that may be) for all elements on the t-level of the combined graph. The small local linear system is assembled and solved using compiler automatic SIMD vectorisation. We use Gaussian Elimination to solve the system directly which seems numerically stable enough and significantly more efficient than a full factorisation. This means the matrix inversion steps are parallelised along the number of nodes in the element which corresponds to vectorising operations on rows of the matrix and along the length of the right-hand side vector.

On the GPU in order to generate sufficient parallel work, the sweep schedules for angles in each angle set (we use eight sets but it is somewhat arbitrary) are combined into a single graph. There is no requirement here that angles in the angle set have identical graphs, rather that their unique graphs are combined into one larger graph respecting all dependencies. As before, parallel work is found through solving all elements on a single t-level of the combined graph for all energy groups. A kernel is launched which, to use terminology from the CUDA programming model, assigns one thread block for each linear system to solve, an approach which is very like that above for the CPU as thread-blocks are assigned to GPU compute units. Each thread-block contains a number of threads which assemble and solve the linear system in parallel. For linear

systems we use 64 threads which corresponds to one thread per matrix entry which greatly improves assembly. For higher orders this is intractable due to GPU device restrictions and so we use one thread per matrix row (or column), an approach similar to the SIMD parallelisation on CPUs however coded explicitly. Hardware atomic memory operations are used to ensure the scalar flux reduction result is correct (recall that there is the potential that more than one angle in a single element is computed in parallel). Manual caching is used to ensure the linear system is assembled and solved in the programmable memory close to each compute unit (shared memory).

As found with the structured case, there is no correlation between the memory bandwidth of the architecture (Fig. 1) and the performance of the transport solve. Therefore, a major performance limiting factor is again due to cache access as the small, dense local matrix is small enough to be cache resident and has high reuse.

3.2 Performance Results

The unstructured sweep code UnSNAP is run on a variety of architectures for first, second and third finite element order mesh. The problem consists of 512 cells, 80 angular directions (organised in 8 angle sets) and 32 energy groups which results in 10.5–83.9 million degrees of freedom (depending on the spatial finite element order). This problem size is representative of the number of unknowns one might typically solve per computational node. The results are shown in Fig. 5 normalised to the Broadwell performance for each element order. As before, all CPUs are dual-socket configurations and we compare to one GPU, whose defining properties are shown in Table II.

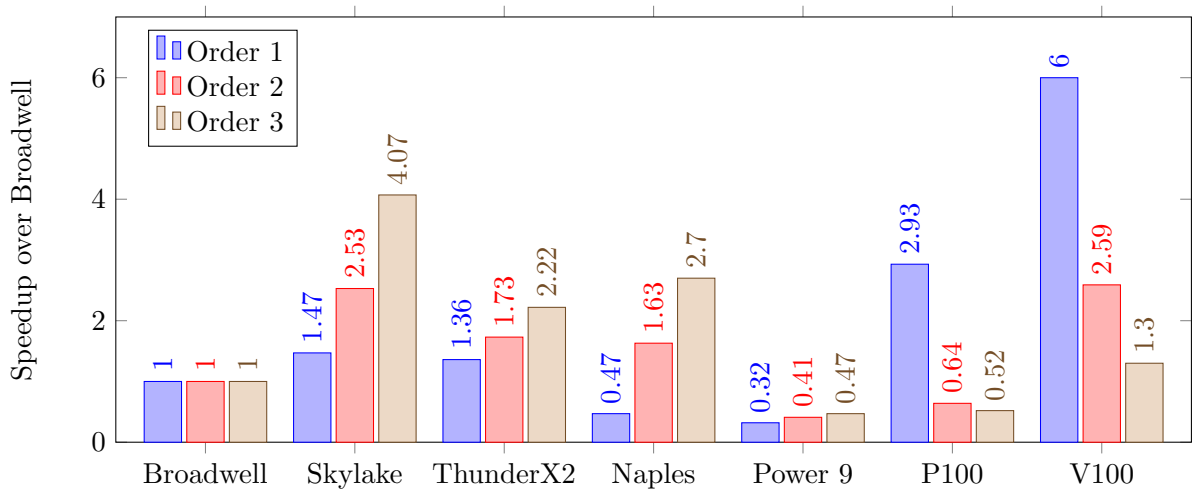

Figure 5: UnSNAP performance results for first, second and third order elements, normalised to Broadwell, augmented from [8].

For linear elements we see that the Intel Skylake and Marvell ThunderX2 (Arm) CPUs perform similarly, around 1.4X over the Intel Broadwell performance. We see the NVIDIA V100 GPU

Table II: NVIDIA GPU details

Processor	Architecture	SMs (GPU compute units)	Main Memory Bandwidth (GB/s)
P100	Pascal	56	732
V100	Volta	80	900

providing 6X improvement for linear elements. The Intel Skylake provides good performance for higher orders indicating that we may need to investigate some implementation optimisations for higher-orders on GPUs. It is important to observe that, as we found with the structured sweeps, the improvements offered by each architecture don't relate to the simple floating-point or main memory bandwidth performance of the processors. Once again, simple performance limiting factors do not apply.

3.3 Processor Performance Bounds

We repeat the experiment described in Section 2.3.1 with the UnSNAP code. This compares the improvement from running the code on one fully populated socket to two half-populated sockets in order to determine the location of the performance limiting factor. The results of this experiment are shown in Fig. 6 which make it clear that no CPU shows significant performance improvement from one to two sockets for any order. As before, despite doing a Gaussian Elimination, the arithmetic intensity is low as more bytes are read for assembly and the solve than number of FLOPs required for the solve. Therefore, UnSNAP is bound by on-core resources for all CPU architectures rather than shared resources, and in particular is not main memory bandwidth bound.

On Broadwell and Power 9 processors, linear order elements show a marked reduction in performance when running under configuration 2. Using the performance profiling tool CrayPAT, we find that the L2 cache hit rate is reduced from 53% to only 7%, thus attributing the performance loss to a significant increase in cache misses.

This can be verified by also examining the cache hit rate on the processors where we find that the L1 cache hit rate is very high indeed, over 95% on Broadwell, for linear elements. Running under configuration 2. in the experiment reduced the L2 cache hit rate significantly, which indicates that access to the close levels of cache dictate the performance of unstructured sweeps on each processor. Indeed, the relative L1 cache bandwidth between the processors does not align with Fig. 5 yet all the CPU L1 caches are large enough to ensure the linear system fits within the cache, and so it is the *access* to cache rather than simply the bandwidth that dictates CPU performance. Caches are designed to inconspicuously improve the performance of applications, and so in depth analysis beyond this level is difficult as the kernel is less tractable to reason about than that of mega-sweep as we did in 2.3.2. Note however that the angular flux is once again much larger than the available last level cache and so must still be read from main memory,

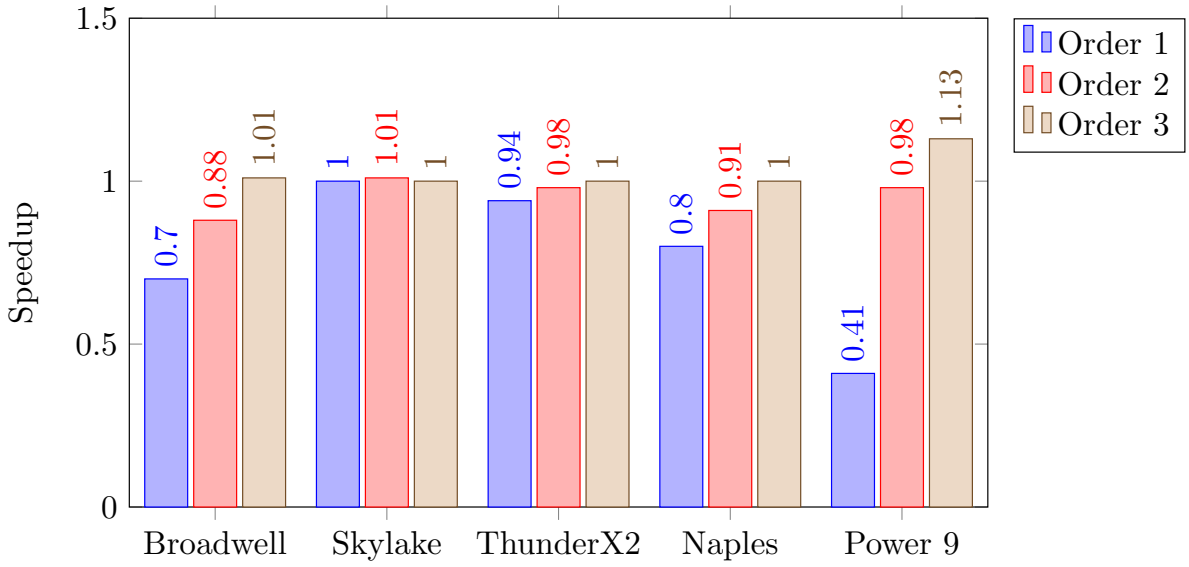

Figure 6: Relative performance improvement of UnSNAP sweeps from Configuration 1. to 2. in the Voysey experiment, augmented from [8].

however the access is infrequent enough when compared to the access to the local linear system and required data for the latter to dominate performance.

Profiling the GPU code shows too that most memory requests are satisfied by the GPU caches rather than the device high-bandwidth memory, a rather unusual performance profile. As for CPUs above, the hit rate we observed for the L2 cache was 96%: very high indeed.

The performance of the access to the data caches on both CPU and GPU architectures is critically important to the speed at which unstructured transport sweeps can be computed on modern many-core processors.

4. CONCLUSIONS

The sweep based algorithms forming a key component of neutron and thermal radiation transport codes will continue to be important on future architectures. This is critical as solving this equation efficiently is challenging. The algorithms and methods used will need to expose sufficient concurrency in order to best exploit the changing landscape of highly parallel computer processors.

It is important that the transport community explore the performance limiting factors of these algorithms to ensure a high level of performance is maintained on many-core processors. This study presents how the performance of the solver kernels at the heart of transport sweeps of both structured and unstructured meshes is limited by certain features of the computing hardware. At scale, the sweep communication often dominates, however it is important to ensure efficient solution of the equations between communications. On many-core CPU architectures, we see that the performance is dictated by properties of the memory cache hierarchy. On

GPUs, structured meshes using a finite difference discretisation are limited by main memory bandwidth as a result of the maximal concurrency exposed, whilst unstructured meshes using a discontinuous Galerkin finite element discretisation are limited by device cache performance instead. This highlights the significant challenges faced by those maintaining a sweep solver over a long period of time.

Although some multi-level approaches are being investigated as an alternative to sweeps (e.g. [27]), these face an equally daunting but significantly different set of challenges.

ACKNOWLEDGEMENTS

This work used the Isambard UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1).

Access to the Cray XC40 supercomputer, Swan, was kindly provided though the Cray Inc. Marketing Partner Network.

Many thanks to Chris Edsall from the ACRC at Bristol University for providing access to AMD Naples on the Oracle Cloud.

Thanks to Rabin Sugumar for assistance with calculating cache performance of mega-sweep on ThunderX2.

We would like to thank the reviewers for their insightful feedback for improvements to this work.

REFERENCES

1. T. Deakin, S. McIntosh-Smith, W. Gaudin. Expressing Parallelism on Many-Core for Deterministic Discrete Ordinates Transport. *Proceedings of the 2015 IEEE International Conference on Cluster Computing.*, Chicago, USA, September 8–11, pp. 729–737 (2015).

2. T. Deakin, S. McIntosh-Smith, M. Martineau, W. Gaudin. An Improved Parallelism Scheme for Deterministic Discrete Ordinates Transport. *International Journal of High Performance Computing Applications.*, **Vol. 32**, Issue 4, pp. 555–569 (2016).

3. T. Deakin, S. McIntosh-Smith, W. Gaudin. Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale. *Proceedings of the 31st International Conference, ISC High Performance.*, Frankfurt, Germany, June 19–23, pp. 429–448 (2016).

4. T. Deakin, W. Gaudin, S. McIntosh-Smith. On the Mitigation of Cache Hostile Memory Access Patterns on Many-Core CPU Architectures. *Proceedings of the 32nd International Conference, ISC High Performance.*, Frankfurt, Germany, June 18–22, pp. 348–362 (2017).

5. T. Deakin, S. McIntosh-Smith, J. Lovegrove, R. Smedley-Stevenson, A. Hagues. UnSNAP: A Mini-App for Exploring the Performance of Deterministic Discrete Ordinates Transport on Unstructured Meshes. *Proceedings of the 2018 IEEE International Conference on Cluster Computing.*, Belfast, Northern Ireland, 10-13 September 10–13, pp. 598–606 (2018).

6. T. Deakin, J. Price, M. Martineau, S. McIntosh Smith. Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream. *International Journal of Computational Science and Engineering*, **Vol 17**, Issue 3, pp. 247–262 (2018).

7. T. Deakin. Leveraging Many-core Technology for Deterministic Neutral Particle Transport at Extreme Scale. *Ph.D. Thesis, University of Bristol*, 2018.

8. T. Deakin, S. McIntosh-Smith, J. Lovegrove, R. Smedley-Stevenson, A. Hagues. Developing a Mini-app for Exploring Algorithms for Unstructured Mesh Deterministic Discrete Ordinates Transport on Many-core Architectures. *Proceedings of the International Conference on Mathematics and Computational Methods (M&C).*, Portland, Oregon, USA, August 25–29, Vol. 2329, pp. 2585–2598 (2019).

9. D. Appelhans, S. Rennich, A. Kunen, L. Grinberg. GPU Optimization of the Kripke Neutron-Particle Transport Mini-App. *Presentation at the GPU Technology Conference (GTC).*, Silicon Valley, California, USA, April 4–8 (2016).

10. R. J. Zerr, R. S. Baker. SNAP: SN (Discrete Ordinates) Application Proxy — Proxy Description. *Los Alamos National Laboratory Report.*, LA-UR-13-21070, 2013.

11. R. S. Baker. An Sn Algorithm for the Massively Parallel CM-200 Computer. *Nuclear Science and Engineering*, **Vol 128**, Number 3, pp. 312–320 (1998).

12. K. R. Koch, R. S. Baker, R. E. Alcouffe. Solution of the first-order form of three-dimensional discrete ordinates equations on a massively parallel machine. *Transactions of the American Nuclear Society*, **Vol 65**, pp. 198–199 (1992).

24. M.P. Adams, M.L. Adams, W.D. Hawkins, T. Smith, L. Rauchwerger, N.M. Amato, T.S. Bailey, R.D. Falgout. Provably Optimal Parallel Transport Sweeps on Regular Grids. *Proceedings of the International Conference on Mathematics and Computational Methods (M&C).*, Sun Valley, Idaho, USA, May 5–9, pp. 2535–2553 (2013).

14. R. S. Baker. An Sn Algorithm for Modern Architectures. *Proceedings of the International Conference on Mathematics and Computational Methods (M&C).*, Nashville, Tennessee, USA, April 19–23, on CD-ROM (2015).

15. A. Kunen, T. Bailey, P. Brown. KRIPKE — A Massively Parallel Transport Mini-app. *Proceedings of the International Conference on Mathematics and Computational Methods (M&C).*, Nashville, Tennessee, USA, April 19–23, on CD-ROM (2015).

16. A. Kunen, J. Loffeld, A. Black, R. Chen, P. Nowak, T. Haut, T. Bailey, P. Brown, S. Rennich, P. Maginot, B. Tagani. Porting 3D Discrete Ordinates Sweep Algorithm in Ardra to CUDA. *Proceedings of the International Conference on Mathematics and Computational Methods (M&C).*, Portland, Oregon, USA, August 25–29, Vol. 2329, pp. 2585–2598 (2019).

17. J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December, pp. 19–25 (1995).

18. S. Rennich, L. Grinberg, A. Kunen. Sn Transport On Accelerators. *Presentation at DOE Centers of Excellence Performance Portability Meeting.*, Glendale, Arizona, USA, April 19–21 (2016).

19. E. Strohmaier, H. Simon, J. Dongarra, M. Meuer. Top 500 — November 2018, http://www.top500.org (2018).

20. M. Feldman. Dennard Scaling Demise Puts Permanent Dent in Supercomputing, https://www.nextplatform.com/2019/06/18/dennard-scaling-demise-puts-permanent-dent-in-supercomputing/ (2019).

21. S. Williams, A. Waterman, D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM.*, **Vol. 52**, pp. 65–76 (2009).

22. A. Voysey, M. Glover. Performance of Met Office Weather and Climate Codes on Cavium ThunderX2 Processors. Presentation at Arm Research Summit, Austin, Texas. https://www.youtube.com/watch?v=xSLY0RJBEAQ (2018).

23. M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, R. Numrich. Improving Performance via Mini-Applications. *Sandia National Laboratories Report.*, SAND2009-5574, pp. 1–38 (2009).

24. M.P. Adams, M.L. Adams, W.D. Hawkins, T. Smith, L. Rauchwerger, N.M. Amato, T.S. Bailey, R.D. Falgout. Provably Optimal Parallel Transport Sweeps on Regular Grids. *Proceedings of the International Conference on Mathematics and Computational Methods (M&C).*, Sun Valley, Idaho, USA, May 5–9, pp. 2535–2553 (2013).

25. T.S. Bailey, R.D. Falgout. Analysis of Massively Parallel Discrete-Ordinates Transport Sweep Algorithms with Collisions. *Proceedings of the International Conference on Mathematics and Computational Methods (M&C).*, Saratoga Springs, New York, USA, May 3–7, pp. 1–5 (2009).

26. W.D. Hawkins, T. Smith, M.P. Adams, L. Rauchwerger, N. Amato, M.L. Adams. Efficient Massively Parallel Transport Sweeps. *Transactions of the American Nuclear Society.*, **Vol. 2017**, pp. 477–481 (2012).

27. R. Smedley-Stevenson. Sweep-Free Deterministic Transport. *Proceedings of the International Conference on Mathematics and Computational Methods (M&C).*, Portland, Oregon, USA, August 25–29, Vol. 2329, pp. 1298–1307 (2019).

28. Y.Y. Azmy. Multiprocessing for Neutron Diffusion and Deterministic Transport Methods. *Progress in Nuclear Engineering*, **Vol 31**, Number 3, pp. 317–368 (1997).